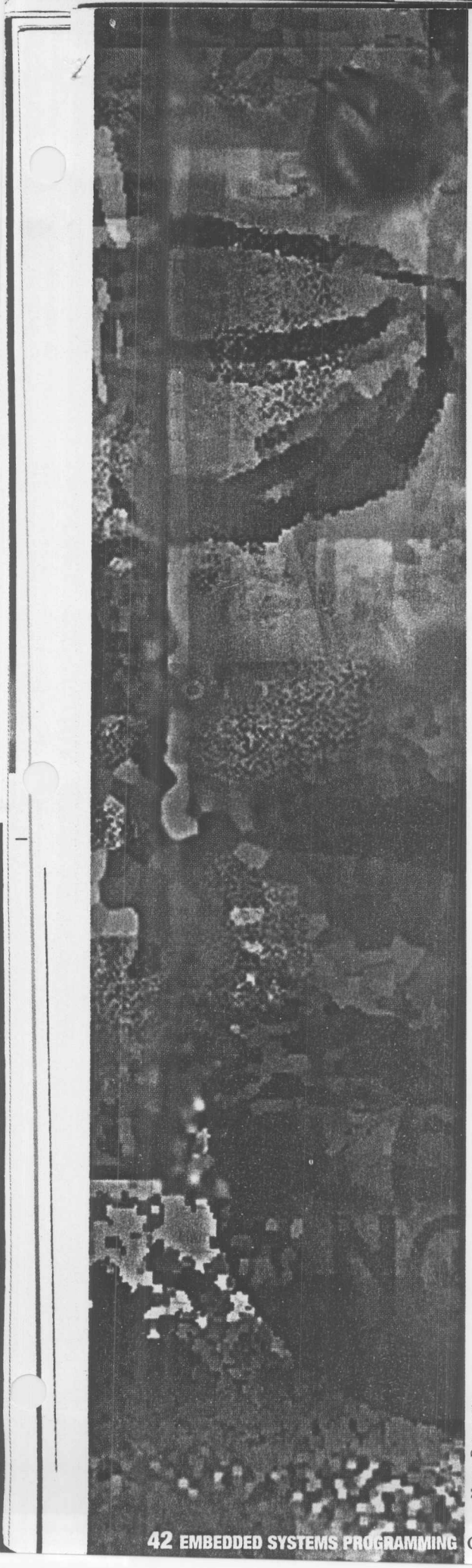


Nance Paternoster

by BRUCE ECKEL

Polymorphism and Virtual Functions in C++

Moving to the highest levels of C++ and object-oriented design means mastering virtual fuctions.

The virtual function is an essential feature of C++ as an object-oriented programming language, along with data abstraction and inheritance. This feature provides another dimension of separation of interface from implementation, to decouple what from how. Virtual functions allow improved code organization as well as the creation of extensible programs that can be "grown" during the original creation of the project, or when new features are desired.

Encapsulation separates the interface from the implementation by hiding the implementation inside a class and making the details private. This sort of mechanical organization makes sense to someone with a procedural programming background. But virtual functions deal with this decoupling in terms of types. Last month, you saw how inheritance allows the treatment of an object as its own type or its base type. This ability is critical because it allows many types derived from the same base type to be treated as if they were one type and a single piece of code to work on all those different types equally. The virtual function allows one type to express its distinction from another, similar type as long as they're both derived from the same base type.

This month, you'll learn about virtual functions starting from the very basics, with simple examples that strip away everything but the "virtualness" of the program.

EVOLUTION OF C++ PROGRAMMERS

C programmers seem to acquire C++ in three steps. First, they think of C++ as simply a better C, since C++ forces you to declare all functions before using them and is much pickier about how variables are used. You can often find the errors in a C program simply by compiling it with a C++ compiler.

The second step is object-based C++. You easily see the code organization benefits of grouping a data structure together with the functions that act on it and the value of constructors and destructors. Most programmers who have been working with C for a while quickly see the usefulness of this because, whenever they create a library, this is exactly what they try to do. With C++, you have the aid of the compiler.

You can get stuck at the object-based level because it's very easy to get to and you get a lot of benefit without much mental effort. It's also easy to feel like you're creating data

types—you make classes and objects and you send messages to those objects, and everything is nice and neat.

Don't be fooled, however. If you stop here, you're missing out on the greatest part of the language, which is the jump to true object-oriented programming. You can only do this with virtual functions.

Since virtual functions manipulate the concept of type rather than just encapsulating code inside structures and behind walls, they are without a doubt the most difficult concept for the new C++ programmer to fathom. However, they're also the turning point in the understanding of object-oriented programming. If you don't use virtual functions, you don't understand object-oriented programming yet.

Because the virtual function is intimately bound with the concept of type, and type is at the core of object-oriented programming, there is no analog to the virtual function in a traditional procedural language. As a procedural programmer, you have no referent with which to think about virtual functions, as you do with almost every other feature in the language. Features in a procedural language can be understood on an algorithmic level, but virtual functions can only be understood from a design viewpoint.

UPCASTING

Last month, you saw how an object can be used as its own type or as an object of its base type. In addition, it can be manipulated through an address of the base type. Taking the address of an object (either a pointer or a reference) and treating it as the address of the base type is called upcasting because of the way inheritance trees are drawn with the base class at the top.

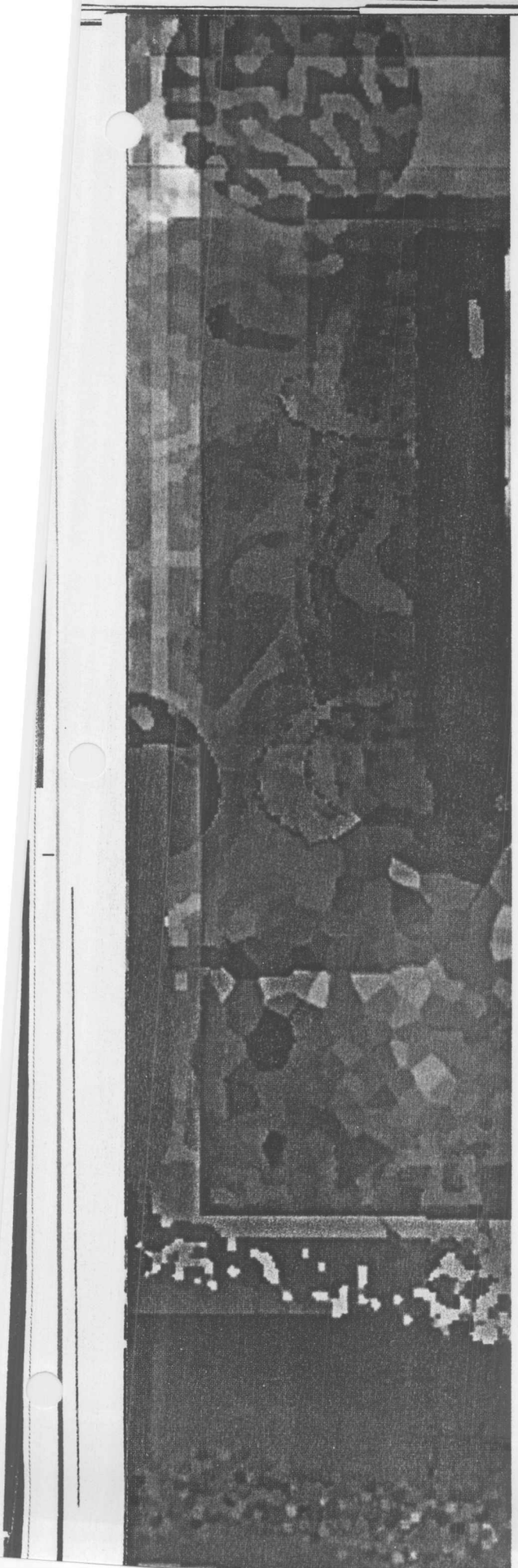
You also saw a problem arise, which is embodied in the following code:

```
//:WIND2.CPP -- inheritance & upcasting
#include <iostream.h>
```

Since virtual functions manipulate the concept of type, they a difficult concept for new C++ programmers.

```
enum note { middleC, Csharp, Cflat }; class
instrument {
public:
  void play(note) {
    cout << "instrument::play" << endl;
  }
};
// wind objects are instruments
// because they have the same interface:
class wind : public instrument {
public:
  // redefine interface function:
  void play(note) {
    cout << "wind::play" << endl;
  }
};
void tune(instrument& i) {
  // ...
  i.play(middleC);
}
main() {
  wind flute;
  tune(flute); // upcasting
}
```

The function `tune()` accepts (by reference) an `instrument`, but also without complaint anything derived from `instrument`. In `main()`, you can see this happening as a `wind` object is passed to `tune()`, with no cast necessary. This is acceptable because the interface in `instrument` must exist in `wind`, since `wind` is publicly inherited from `instrument`. Upcasting from `wind` to `instrument` may "narrow" that interface, but it cannot



Polymorphism and Virtual Functions

make it any less that the full interface to `instrument`.

The same arguments are true when dealing with pointers; the only difference is that the user must explicitly take the addresses of objects as they are passed into the function.

THE PROBLEM

The problem with WIND2.CPP can be seen by running the program. The output is `instrument::play`. This is clearly not the desired output, since you happen to know that the object is actually a `wind` and not just an `instrument`. The desired behavior is for `wind::play` to be called. For that matter, any object of a class derived from `instrument` should have its version of `play` used, regardless of the situation.

However, the behavior of WIND2.CPP is not surprising given C's approach to functions. To understand the issues, you need to be aware of the concept of binding.

Connecting a function call to a function body is called binding. When binding is performed before the program is run (by the compiler and linker) it's called early binding. You may not have heard the term before because it's never been an option with procedural languages: C compilers only have one kind of function call, and that's early binding.

The problem in the previous program is caused by early binding, since the compiler cannot know the correct function to call when it only has an `instrument` address.

The solution is called late binding. The binding occurs at run time, based on the type of object. When a language implements late binding, there must be some mechanism to determine the type of the object at run time and call the appropriate member function. That is, the compiler still doesn't know what the actual object type is but it inserts code that finds out and makes the right call. The late-binding mechanism varies from language to language but you can imagine that some sort of type information must be installed in the objects themselves. You'll see how this works later.

VIRTUAL FUNCTIONS

To cause late binding to occur for a particular function, C++ requires that you use the `virtual` keyword when declaring the function in the base class. Late binding only occurs with `virtual` functions and only when you're using an address of the base class where those `virtual` functions exist, although they may also be defined in an earlier base class.

To create a member function as `virtual`, you simply precede the declaration of the function with the keyword `virtual`. You don't repeat it for the function definition, and you don't need to repeat it in any of the derived-class function redefinitions, though it does no harm to do so. If a function is declared as `virtual` in the base class, it is `virtual` in all the derived classes.

To get the desired behavior from WIND2.CPP, simply add the `virtual` keyword in the base class before `play()`:

```
//:WIND3.CPP -- Late binding with virtual
#include <iostream.h>
enum note { middleC, Csharp, Cflat }; class
instrument {
public:
  virtual void play(note) {
    cout << "instrument::play" << endl;
  }
};
// wind objects are instruments
// because they have the same interface:
class wind : public instrument {
public:
  // redefine interface function:
  void play(note) {
    cout << "wind::play" << endl;
  }
};
void tune(instrument& i) {
  // ...
  i.play(middleC);
}
main() {
  wind flute;
```

```
  tune(flute); // upcasting
}
```

This file is identical to WIND2.CPP except for the addition of the `virtual` keyword, and yet the behavior is significantly different: Now the output is `wind::play`.

EXTENSIBILITY

With `play()` defined as `virtual` in the base class, you can add new types to the system without changing the `tune()` function. In a well-designed object-oriented program, most or all your functions will follow the model of `tune()` and only communicate with the base-class interface. Such a program is extensible, because you can add new functionality simply by inheriting new data types from the common base class. No functions that manipulate the base-class interface will need to be changed to accommodate the new classes.

Listing 1 shows the instrument example with more virtual functions and a number of new classes, all of which work correctly with the old, unchanged `tune()` function.

You can see that another inheritance level has been added beneath `wind`, but the `virtual` mechanism works correctly no matter how many levels there are. The `adjust()` function is not redefined for `brass` and `woodwind`. When this happens, the previous definition is automatically used; the compiler guarantees there's always some definition for a `virtual` function, so you'll never end up with a call that doesn't bind to a function body. That situation would spell disaster.

The array `A[]` contains pointers to the base class instrument, so upcasting occurs during the process of array initialization. This array and the function `f()` will be used in later discussions.

In the call to `tune()`, upcasting is performed on each different type of object, and yet the desired behavior always takes place. This can be described as "sending a message to an object and letting the object worry about what to do with it." The `virtual` function is the lens you should use when you're trying to analyze a project: Where should the base class occur, and how might you want to extend the program?

However, even if you don't discover the proper base class interfaces and virtual functions at the initial creation of the program, you'll often discover

LISTING 1
WIND4.CPP

```
//:WIND4.CPP -- Extensibility in OOP
#include <iostream.h>
enum note { middleC, Csharp, Cflat };
// etc.

class instrument {
public:
  virtual void play(note) {
    cout << "instrument::play" << endl;
  }
  virtual char* what() {
  return "instrument"; }
  virtual void adjust(int) {}
};

class wind : public instrument {
public:
  void play(note) {
        cout << "wind::play" << endl;
  }
  virtual char* what() {
  return "wind"; }
  virtual void adjust(int) {}
};

class percussion : public instrument {
public:
  void play(note) {
    cout << "percussion::play" << endl;
  }
  virtual char* what() {
  return "percussion"; }
  virtual void adjust(int) {}
};

class string : public instrument {
public:
  void play(note) {
    cout << "string::play" << endl;
  }
  virtual char* what() {
  return "string"; }
  virtual void adjust(int) {}
};

class brass : public wind {
public:
  void play(note) {
    cout << "brass::play" << endl;
  }
  virtual char* what() {
  return "brass"; }
};

class woodwind : public wind {
public:
  void play(note) {
    cout << "woodwind::play" << endl;
  }
  virtual char* what() {
  return "woodwind"; }
};

// identical function from before:
void tune(instrument& i) {
  // ...
  i.play(middleC);
}

// new function:
void f(instrument& i) { i.adjust(1); }

// upcasting during array
// initialization:
instrument* A[] = {
  new wind,
  new percussion,
  new string,
  new brass
};

main() {
  wind flute;
  percussion drum;
  string violin;
  brass flugelhorn;
  woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
}
```

them later, even much later, when you set out to extend or otherwise maintain the program. This is not an analysis or design error; it simply means you didn't have all the information the first time. Because of the tight class modularization in C++, it isn't a large problem when this occurs because changes you make in one part of a system tend not to propagate to other parts of the system as they do in C.

HOW C++ IMPLEMENTS LATE BINDING

How can late binding happen? All the work goes on under the covers by the compiler, which installs the necessary late binding mechanism when you ask it to by creating virtual functions. Since programmers often benefit from understanding the mechanism of virtual functions in C++, this section will elaborate on the way the compiler implements this mechanism.

The keyword **virtual** tells the compiler it should not perform early binding. Instead, it should automatically install all the mechanisms necessary to perform late binding. If you call **play()** for a **brass** object through an address for the base class **instrument**, you'll get the proper function.

To accomplish this, the compiler creates a single table, called the **VTABLE**, for each class that contains **virtual** functions. The compiler places the addresses of the virtual functions for that particular class in the **VTABLE**. In each class with virtual functions, it secretly places a pointer, called the **vpointer** (or **VPTR**) that points to the **VTABLE** for that object. When you make a virtual function call through a base-class pointer (a polymorphic call), the compiler quietly inserts code to fetch the **VPTR** and look up the function address in the **VTABLE**, thus calling the right function and causing late binding to take place.

All of this—setting up the **VTABLE** for each class, initializing the **VPTR**, inserting the code for the virtual function call—happens automatically, so you don't have to worry about it. With virtual functions, the proper function gets called for an object, even if the compiler cannot know the specific type of the object.

STORING TYPE INFORMATION

You can see that there is no explicit type information stored in any of the classes. But the previous examples and simple logic tell you that there must be some sort of type information stored in the objects, otherwise the type could not be established at run time. This is true, but the type information is hidden. To see it, the following is an example that examines the sizes of classes that use virtual functions compared with those that don't:

```
//: SIZES.CPP -- Object sizes vs.
//  virtual funcs
#include <iostream.h>
class no_virtual {
  int a;
public:
  void x() {}
  int i() { return 1; }
};
class one_virtual {
  int a;
public:
  virtual void x() {}
  int i() { return 1; }
};
class two_virtuals {
  int a;
public:
  virtual void x() {}
  virtual int i() { return 1; }
};
void main() {
  cout << "int: " << sizeof(int) << endl;
  cout << "no_virtual: "
       << sizeof(no_virtual) << endl;
  cout << "void* : " << sizeof(void*)
       << endl;
  cout << "one_virtual: "
       << sizeof(one_virtual) << endl;
  cout << "two_virtuals: "
       << sizeof(two_virtuals) << endl;
}
```

With no virtual functions, the size of the object is exactly what you'd expect: the size of a single **int**. With a single virtual function in **one_virtual**, the size of the object is the size of **no_virtual** plus the size of a **void** pointer. It turns out that the compiler inserts a single pointer (the **VPTR**) into the structure if you have one or more virtual functions. There is no size difference between **one_virtual** and **two_virtuals**. That's because the **VPTR** points to a table of function addresses. You only need one, since all the virtual function addresses are contained in that single table.

This example required at least one data member. If there had been no data members, the C++ compiler would have forced the objects to be a nonzero size because each object must have a distinct address. If you imagine indexing into an array of zero-sized objects, you'll understand. A dummy member is inserted into objects that would otherwise be zero-sized. When the type information is inserted because of the **virtual** keyword, this takes the place of the dummy member. Try commenting out the **int a** in all the classes in the previous example to see this.

PICTURING VIRTUAL FUNCTIONS

To understand exactly what's going on when you use a virtual function, it's helpful to be able to imagine the activities going on under the covers. Figure 1 shows a drawing of the array of pointers **A[]** in WIND4.CPP.

The array of **instrument** pointers has no specific type information; they each point to an object of type **instrument**. Since **wind**, **percussion**, **string** and **brass** are derived from **instrument**, they all fit into this category, and have the same interface as **instrument**, and can respond to the same messages. Their addresses can also be placed into the array. However, the compiler doesn't know they are anything more than **instrument** objects, so left to its own devices, it would normally call the base-class versions of all the functions. But in this case, all those functions

have been declared virtual, so something different happens.

Each time you create a class that contains virtual functions, or you derive a class from a class that contains virtual functions, the compiler creates a VTABLE for that class, seen in Figure 1. In that table, it places the addresses of all the functions that are declared virtual in this class or in the base class. If you don't redefine a function that was declared virtual in the base class, the compiler simply uses the address of the base-class version in the derived class (you can see this in the adjust entry in the brass VTABLE). Then it places the VPTR (discovered in SIZES.CPP) into the class. There is only one VPTR for each object when using simple inheritance like this. The VPTR must be initialized to point to the starting address of the appropriate VTABLE (this happens in the constructor, which you'll see later in more detail).

Once the VPTR is initialized to the proper VTABLE, the object in effect "knows" what type it is. But this self-knowledge is worthless unless it is used at the point a virtual function is called.

When you call a virtual function through a base class address (when the compiler doesn't have all the information necessary to perform early binding), something special happens. Instead of performing a typical function call, which is simply an assembly-language CALL to a particular address, the compiler generates different code to perform the function call. Figure 2 shows what a call to adjust() for a brass object it looks like if made through an instrument pointer (an instrument reference produces the same result).

The compiler starts with the instrument pointer, which points to the starting address of the object. All instrument objects, or objects derived from instrument, have their VPTR in the same place (often at the beginning of the object), so the compiler can pick the VPTR out of the object. The VPTR points to the starting address of the VTABLE. All the VTABLEs are laid out in the same order, regardless of the specific type of the object. play() is first, what() is second, and adjust() is third. The compiler knows that regardless of the specific object type, the adjust() function is at the location VPTR+2. Thus, instead of saying "call the function at the absolute location instrument::adjust" (early binding: the wrong action), it generates code that says, in effect, "call the function at VPTR+2." Since the fetching of the VPTR and the determination of the actual function address occurs at run time, you get the desired late binding. You send a message to the object, and the object figures out what to do with it.

UNDER THE HOOD

It can be helpful to see the assembly language code generated by a virtual function call, so you can see that late-binding is indeed taking place. Here's the output from one compiler (Borland C++ v. 4.02, small model) for the call:

```
i.adjust(1):
```

inside the function f(instrument& i):

```
push  1
push  si
mov   bx,word ptr [si]
call  word ptr [bx+4]
add   sp,4
```

The arguments of a C++ function call, like a C function call, are pushed on the stack from right to left, so the argument 1 is pushed on the stack first. At this point in the function, the register si (part of the Intel X86 processor architecture) contains the address of i. This is also pushed on the stack because it is the starting address of the object of interest. Remember that the starting address corresponds to the value of this, and this is quietly

FIGURE 1
Late-binding structure.

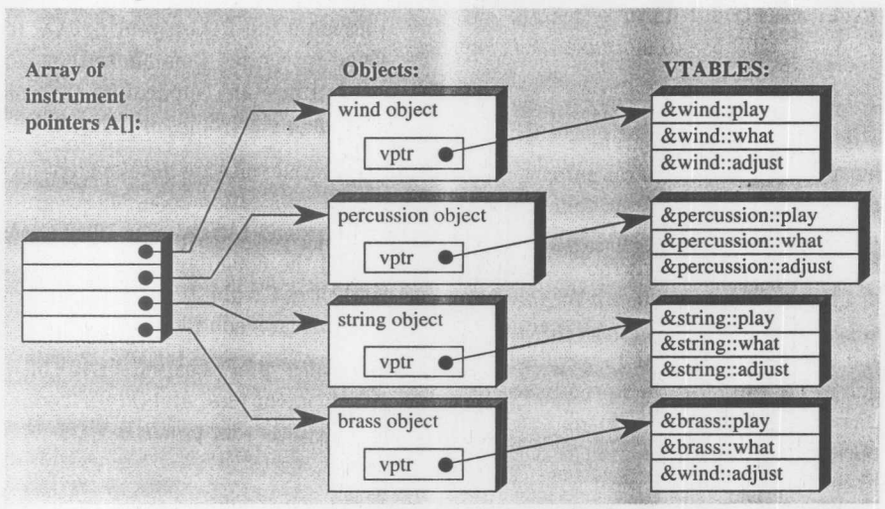
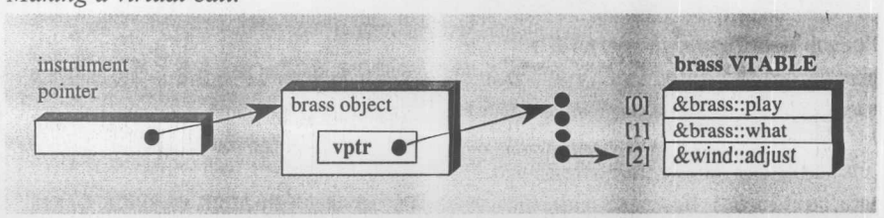


FIGURE 2
Making a virtual call.



pushed on the stack as an argument before every member function call, so the member function knows which particular object it is working on. Thus, you'll always see the number of arguments plus one pushed on the stack before a member function call (except for `static` member functions, which have no `this`).

Now the actual virtual function call must be performed. First, the `VPTR` must be produced, so the `VTABLE` can be found. For this compiler the `VPTR` is inserted at the beginning of the object, so the contents of `this` correspond to the `VPTR`. The line:

```
mov  bx,word ptr [si]
```

fetches the word that `si` (that is, `this`) points to, which is the `VPTR`. It places the `VPTR` into the register `bx`.

The `VPTR` contained in `bx` points to the starting address of the `VTABLE`, but the function pointer to call isn't at the zeroth location of the `VTABLE`, but the second location, since it's the third function in the list. For this memory model, each function pointer is two bytes long, so the compiler adds four to the `VPTR` to calculate where the address of the proper function is.

This is a constant value established at compile time, so the only thing that matters is that the function pointer at location number two is the one for `adjust()`. Fortunately, the compiler takes care of all the bookkeeping for you and ensures that all the function pointers in all the `VTABLE`s occur in the same order.

Once the address of the proper function pointer in the `VTABLE` is calculated, that function is called. So the address is fetched and called all at once in the statement:

```
call  word ptr [bx+4]
```

Finally, the stack pointer is moved to clean off the arguments that were pushed before the call. (In C++ the caller is responsible for cleaning off the arguments.) In some languages, like Pascal, the function code itself cleans off the arguments.

INSTALLING THE VPOINTER

Since the `VPTR` determines the virtual function behavior of the object, it's critical that the `VPTR` always point to the proper `VTABLE`. You don't ever want to be able to make a call to a virtual function before the `VPTR` is properly initialized. Of course, the place where initialization can be guaranteed is in the constructor, but no WIND examples have a constructor.

This is where the default creation of constructors is essential. In the WIND examples, a default constructor is created by the compiler that does nothing except initialize the `VPTR`. This con-

structor, of course, is automatically called for all `instrument` objects before you can do anything with them, so you know that it's always safe to call virtual functions.

The implications of the automatic initialization of the `VPTR` inside the constructor are discussed in a later section.

OBJECTS ARE DIFFERENT

Upcasting only deals with addresses. If the compiler has an object, it knows the exact type and therefore (in C++) will not use late binding for any function calls—or at least, the compiler doesn't need to use late binding. For efficiency's sake, most compilers will perform early binding when they are making a call to a virtual function for an object. Here's an example:

```
//:EARLY.CPP -- early binding & virtuals
#include <iostream.h>
class base {
public:
  virtual int f() { return 1; }
};
class derived : public base {
public:
  int f() { return 2; }
};
main() {
  derived d;
  base * b1 = &d;
  base & b2 = d;
  base b3;
  // late binding for both:
  cout << "b1->f() = " << b1->f() << endl;
  cout << "b2.f() = " << b2.f() << endl;
  // early binding (probably):
  cout << "b3.f() = " << b3.f() << endl;
}
```

In `b1->f()` and `b2.f()`, addresses are used. The information is incomplete: `b1` and `b2` can represent the address of a base or something derived from a base, so the virtual mechanism must be used. When calling `b3.f()`, there's no ambiguity. The compiler knows the exact type and that it's an object, so it can't possibly be an object derived from `base`, it's exactly a `base`. Thus, early binding is probably used. However, if the compiler doesn't want to work so hard, it can still use late binding and the same behavior will occur.

If this technique is so important, and if it makes the right function call all the time, why is it an option? Why do you even need to know about it? The answer is part of the fundamental philosophy of C++: because it's not quite as efficient. You can see from the previous assembly language output that instead of one simple `CALL` to an absolute address, two more sophisticated assembly instructions are required to set up the virtual function call. This requires code space and execution time.

Some object-oriented languages have taken the approach that late binding is so intrinsic to object-oriented programming that it should always take place, that it should not be an option, and the user shouldn't have to know about it. This is a design decision when creating a language, and that particular path is appropriate for many languages. However, C++ comes from the C heritage, where efficiency is critical. After all, the original reason C was created was to replace assembly language for the implementation of an operating system (thereby rendering that operating system—UNIX—far more portable than its predecessors).

One of the main reasons for the invention of C++ was simply to make C programmers more efficient. And the first question asked when C programmers encounter C++ is "what kind of size and speed impact will I get?" If the answer was "everything's great except for function calls when you'll always have a little extra overhead," many people would stick with C rather than making the change to C++. Thus, the virtual function is an option, and the language defaults to nonvirtual, which is the fastest configuration. Bjarne Stroustrup stated that his guideline was "if you don't use it, you don't pay for it."

Anecdotal evidence suggests that the size and speed impacts of going to C++ are within 10% of the size and speed of C and often much closer to the same. The reason you might get greater size and speed efficiency is because you may design a C++ program in a smaller, faster way than you would using C.

ABSTRACT BASE CLASSES

In all the instrument examples, the functions in the base class `instrument` were always dummy functions. If these functions were ever called, they indicate you've done something wrong. The intent of `instrument` is only to create a common interface for all the classes derived from it, as shown in Figure 3.

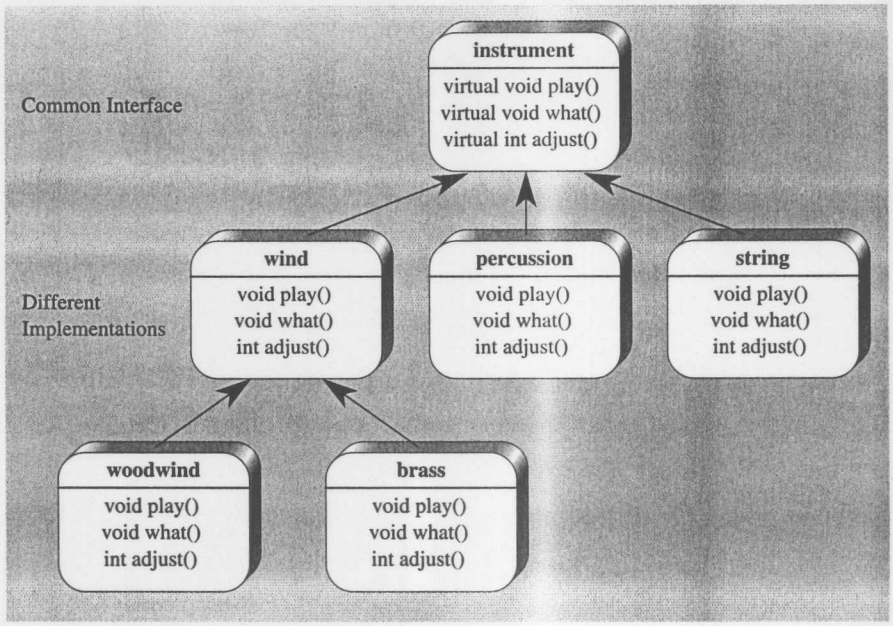
The dashed lines indicate a class and suggest its nonphysical nature, since a class is only a description and not a physical item. The arrows from the derived classes to the base class indicate the inheritance relationship.

The only reason to establish the common interface is so it can be expressed differently for each different subtype. It's a basic form to use, so you can say what's in common with all the derived classes. Nothing else. Another way of saying this is to call `instrument` an abstract base class, or simply an abstract class. You create an abstract class when you want to manipulate a set of classes through this common interface.

You are only required to declare a function as `virtual` in the base class. All derived-class functions that match the signature of the base-class declaration will be called using the virtual mechanism. Some people use the `virtual` keyword in the derived-class declarations for clarity, but it is redundant.

If you have a genuine abstract class, such as `instrument`, objects of that class almost always have no meaning. That is, since `instrument` is only meant to express the interface and not a particular implementation, creating an `instrument` object makes no sense, and you'll probably want to prevent the user from doing it. This can be accomplished by making all the virtual functions in

FIGURE 3
Instrument class hierarchy.



`instrument` print error messages, but this delays the information until run time, and requires reliable exhaustive testing on the part of the user. It is much better to catch the problem at compile time.

C++ provides a mechanism for doing this called the pure virtual function. Following is the syntax used for a declaration:

```
virtual void X() = 0;
```

In effect, you are assigning the function body to zero or telling the compiler "there is no function body for this." If there is no function body, there can be no address in the VTABLE. In fact, if only one function in a class is declared as pure virtual, the compiler will not generate a VTABLE for that class. A class containing pure virtual functions is called a pure abstract base class.

If there is no VTABLE for a class, what will the compiler do when someone tries to make an object of that class? It looks around to see what the VTABLE address is, so it can generate code inside the constructor to stick that address into the VPTR. However, there is no VTABLE for a pure abstract class, so the compiler gives you a message saying you can't create objects of a pure abstract class. Thus, the compiler insures the purity of the abstract class, and you don't have to worry about misusing it.

Here's WIND4.CPP modified to use pure virtual functions:

```
//:WIND5.CPP -- Pure abstract base classes
#include <iostream.h>
enum note { middleC, Csharp, Cflat };class
instrument {
public:
  // pure virtual:
  virtual void play(note) = 0;    virtual
char* what() = 0;
  virtual void adjust(int) = 0;
};
// rest of the file is the same ...
```

Pure virtual functions are very helpful because they make explicit the abstractness of a class and tell the user and compiler how it was intended to be used.

Just because one pure virtual function prevents the VTABLE from being generated doesn't mean you don't want function bodies for some of the others. Often, you will want to call a base-class version of a function, even if the version is virtual. It's always a good idea to try to put as much common code as possible into classes as close to the root of your hierarchy as possible. Not only does this save code space, it allows easy propagation of changes.

INHERITANCE AND THE VTABLE

You can imagine what happens when you perform inheritance and redefine some of the virtual functions. The compiler creates a new VTABLE for your new class, and it inserts your new function addresses, using the base-class function addresses for any virtual functions you don't redefine. One way or another, there's always a full set of function addresses in the VTABLE, so you'll never be able to make a call to an address that isn't there (which would be disastrous).

But what happens when you inherit and add new virtual functions in the derived class? Following is a simple example:

```
//: ADDV.CPP--adding virtuals in derivation
#include <iostream.h>
class base {
  int i;
public:
  base(int I) : i(I) {}
  virtual int value() { return i; }
};
class derived : public base {
public:
  derived(int I) : base(I) {}
  int value() { return base::value() * 2; }
  // New virtual function in derived class:
  virtual int shift(int x) {
    return base::value() << x;
  }
};
void main() {
  base* B[] = { new base(7), new derived(7)
};
  cout << "B[0]->value() = "
       << B[0]->value() << endl;
  cout << "B[1]->value() = "
       << B[1]->value() << endl;
//  cout << "B[1]->shift(3) = "
//       << B[1]->shift(3) << endl;
// illegal
}
```

The class **base** contains a single virtual function **value()**, and **derived** adds a second one called **shift()** as well as redefining the meaning of **value()**. A diagram will help visualize what's happening. Figure 4 shows the **VTABLE**s that are created by the compiler for **base** and **derived**.

The compiler maps the location of the **value** address into exactly the same spot in the **derived VTABLE** as it is in the **base VTABLE**. Similarly, if a class is inherited from **derived**, its version of **shift** would be placed in its **VTABLE** in exactly the same spot as it is in **derived**. As you saw with the assembly language example, the compiler generates code that uses a simple numerical offset in the **VTABLE** to select the virtual function. Regardless of what specific subtype the object belongs to, its **VTABLE** is laid out the same way, so calls to the virtual functions will always be made the same way.

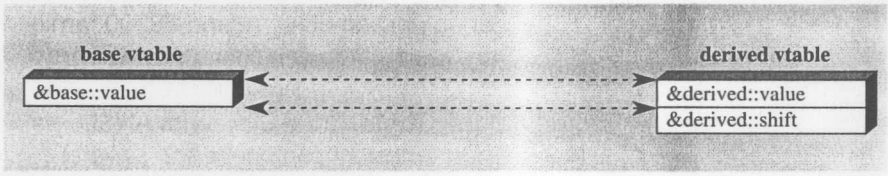
In this case, however, the compiler is only working with a pointer to a base-class object. The base class only has the **value()** function, so that is the only function the compiler will allow you to call. How could it possibly know that you are working with a **derived** object if it only has a pointer to a base-class object? That pointer might point to some other type that doesn't have a **shift** function. It may or may not have some other function address at that point in the **VTABLE**, but in either case, making a virtual call to that **VTABLE** address is not what you want to do. So it's fortunate and logical that the compiler protects you from making virtual calls to functions that only exist in derived classes.

There are some odd cases where you may know that the pointer actually points to an object of a specific subclass. If you want to call a function that only exists in that subclass, you must cast the pointer. You can repair the error in the previous program like this:

```
((derived *)B[1])->shift(3)
```

Here, you happen to know that **B[1]** points to a **derived** object, but in the general case, you don't know that. If

FIGURE 4
Adding virtuals in derived classes.



your problem is set up so you must know the exact types of all objects, you should rethink it because you're probably not using virtual functions properly. However, there are some situations where the design works best if you know the exact type of all objects kept in a generic container. This is the problem of run-time type identification.

Run-time type identification is all about casting pointers to base class objects down to pointers to derived class objects. (Up and down are relative to a typical class diagram, with the base class at the top.) Casting up happens automatically with no coercion because it's completely safe. Casting down is unsafe, since there's no compile time information about the actual types, so you must know exactly what type the object really is. If you cast it into the wrong type, you will be in trouble.

OBJECT SLICING

There is a distinct difference between passing addresses and passing values when treating objects polymorphically. All the examples you've seen here, and virtually all the examples you will see, pass addresses and not values. Addresses all have the same size, so passing the address of an object of a derived type (which is usually bigger) is the same as passing the address of an object of the base type (which is usually smaller). As explained before, the goal when using polymorphism is code that manipulates objects of a base type and can also transparently manipulate derived-type objects.

When passing by value, the situation changes. Here's an example to illustrate the problem:

```
//: SLICE.CPP -- Object slicing
#include <iostream.h>
class base {
  int i;
public:
  base(int I = 0) : i(I) {}
  virtual int sum() { return i; }
};
class derived : public base {
  int j;
public:
  derived(int I = 0, int J = 0)
    : base(I), j(J) {}
  int sum() { return base::sum() + j; }
};
void call(base b) {
  cout << "sum = " << b.sum() << endl;
}

void main() {
  base b(10);
  derived d(10, 47);
  call(b);
  call(d);
}
```

The function `call()` is passed an object of type `base` by value. It then calls the virtual function `sum()` for the `base` object. In `main()`, you might expect the first call to produce 10 and the second to produce 57. In fact, both calls produce 10.

Two things are happening in this program. First, since `call()` only accepts a `base` object, all the code inside the function body will only manipulate members associated with `base`. Any calls to `call()` will only cause an object the size of `base` to be pushed on the stack and cleaned up after the call. If an object of a class derived from `base` is passed to call(), the compiler accepts it, but it only pushes the `base` portion of the object on the stack. It slices the derived portion off of the object, as shown in Figure 5.

You may wonder about the virtual function call. Here, the virtual function makes use of portions of `base` (which still exists) and `derived`, which no longer exists because it was sliced off.

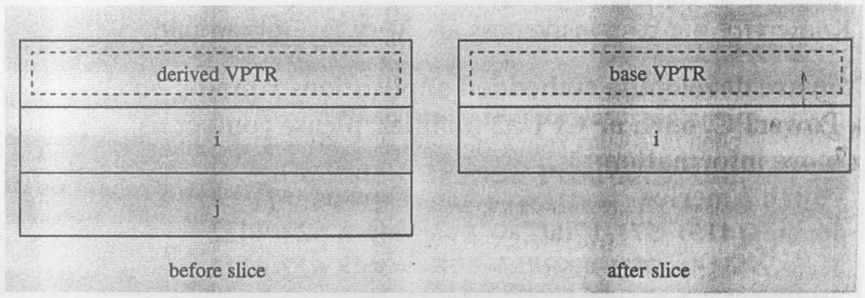
So what happens when the virtual function is called? You're saved from disaster precisely because the object is being passed by value. The compiler thinks it knows the precise type of the object. Here, it does, because any information that contributed extra features to the objects has been lost. To pass by value, it uses the copy constructor for a `base` object, which initializes the `VPTR` to the `base` `VTABLE` and only copies the `base` parts of the object. There's no explicit copy constructor here, so the compiler synthesizes one. Under all interpretations, the object truly becomes a `base` during slicing.

CONSTRUCTORS

When an object containing virtual functions is created, its `VPTR` must be initialized to point to the proper `VTABLE`. This must be done before there's any possibility of calling a virtual function. Since the constructor has the job of bringing an object into existence, it is also the constructor's job to set up the `VPTR`. The compiler secretly inserts code into the beginning of the constructor that initializes the `VPTR`. Even if you don't explicitly create a constructor for a class, the compiler will create one for you with the proper `VPTR` initialization code (if you have virtual functions). This has several implications.

The first concerns efficiency. One of the prime reasons for `inline` functions is to reduce the calling overhead for

FIGURE 5
Object slicing.



small functions. If C++ didn't provide `inline` functions, the preprocessor might be used to create these macros. However, the preprocessor has no concept of access or classes, and therefore couldn't be used to create member function macros. In addition, with constructors that must have hidden code inserted by the compiler, a preprocessor macro wouldn't work at all.

You must be aware when hunting for efficiency holes that the compiler is inserting hidden code into your constructor function. Not only must the constructor initialize the `VPTR`, but it must also insure that the `this` pointer is nonzero (for the heap objects). Taken together, this code can impact what appeared to be a tiny inline function call. In particular, the size of the constructor can overwhelm the savings you get from reduced function-call overhead. If you make a lot of inline constructor calls, your code size can grow without any benefits in speed.

Of course, you probably won't make all tiny constructors non-inline right away because they're much easier to write as inlines. But remember when you're tuning your code to remove inline constructors.

ORDER OF CONSTRUCTOR CALLS

The second interesting facet of constructors and virtual functions concerns the order of constructor calls and the way virtual calls are made within constructors. Consider the following example that illustrates the order of constructor calls when using inheritance:

```
//: ORDER.CPP -- Order of constructor calls
//. with inheritance
#include <iostream.h>
#define inherit(derived, base) \
class derived : public base { \
public: \
  derived() { cout << #derived << endl; } \
};
class X {};
inherit(A, X)
inherit(B, A)
inherit(C, B)
void main() { C c; }
```

A macro was used here to generate the code automatically. The base `class X` is a dummy class that simply provides something to inherit from when creating `class A`.

If the constructor were an ordinary member function, only the local version of the function would be called. You know the constructor isn't ordinary when you see the output of this program: `A B C`. All the base class constructors are always called during inheritance. This makes sense because the constructor has a special job: to see that the object is built properly. Since a derived class only has access to its own members and not those of the base class, only the base class constructor can properly initialize its own elements. Therefore it's essential that all constructors get called, otherwise the entire object wouldn't be constructed properly. That's why the compiler enforces a constructor call for every portion of a derived class. It will call the default constructor if you don't explicitly call a base-class constructor in the constructor initializer list. If there is no default constructor, the compiler will complain. (In this example, `class X` has no constructors, so the compiler can automatically make a default constructor.)

The order of the constructor calls is important. When you inherit, you know all about the base class and can access any public and protected members of the base class. You must be able to assume that all the members of the base class are valid when you're in the derived class. In a normal member function, construction has already taken place, so all the members of all parts of the object have been built. Inside the constructor, however, you must be able to assume that all mem-

bers that you use have been built. The only way to guarantee this is for the base-class constructor to be called first. Then, when you're in the derived-class constructor, all the members you can access in the base class have been initialized. Knowing all members are valid inside the constructor is also the reason that, whenever possible, you should initialize all member objects (objects placed in the class using composition) in the constructor initializer list. If you follow this practice, you can assume all base class members and member objects of the current object have been initialized.

BEHAVIOR OF VIRTUAL FUNCTIONS

The hierarchy of constructor calls brings up an interesting dilemma. What happens if you're inside a constructor and you call a virtual function? Inside an ordinary member function you can imagine what will happen—the virtual call is resolved at run time because the object cannot know whether it belongs to the class the member function is in or some class derived from it. For consistency, you might think this is what should happen inside constructors.

This is not the case. If you call a virtual function inside a constructor, only the local version of the function is used. That is, the virtual mechanism doesn't work within the constructor.

This behavior makes sense for two reasons. Conceptually, the constructor's job is to bring the object into existence—hardly an ordinary feat. Inside any constructor, the object may only be partially formed. You can only know that the base-class objects have been initialized, but you cannot know what classes are inherited from you. A virtual function call, however, reaches forward or outward into the inheritance hierarchy. It calls a function in a derived class. If you could do this inside a constructor, you'd be calling a function that might manipulate members not yet initialized, a sure recipe for disaster.

The second reason is a mechanical one. When a constructor is called, one of the first things it does is initialize its `VPTR`. However, it can only know that it is of the "current" type. The constructor code is completely ignorant of whether the object is in the base of another class. When the compiler generates code for that constructor, it generates code for a constructor of that class, not a base class or a class derived from it. The `VPTR` it uses must be for the `VTABLE` of that class. The `VPTR` remains initialized to that `VTABLE` for the rest of the object's lifetime unless this isn't the last constructor call. If a more-derived constructor is called afterward, that constructor sets the `VPTR` to its `VTABLE` and so on, until the last constructor finishes. The state of the `VPTR` is determined by which constructor is called last. This is another reason why the constructors are called in order, from base to most-derived.

But while this series of constructor calls is taking place, each constructor has set the `VPTR` to its own `VTABLE`. If it uses the virtual mechanism for function calls, it will only produce a call through its own `VTABLE`, not the most-derived `VTABLE`, as would be the case after all the constructors were called. Many compilers recognize that a virtual function call is being made inside a constructor, and perform early binding because they know that late binding will only produce a call to the local function. In either event, you won't get the results you might expect from a virtual function call inside a constructor.

VIRTUAL DESTRUCTORS

While constructors cannot be made explicitly `virtual`, destructors can and often must be made `virtual` so they will operate properly.

The constructor has the special job of putting an object together piece by

piece, first by calling the base constructor, then the more derived constructors in order. Similarly, the destructor also has a special job. It must disassemble an object that may belong to a hierarchy of classes. To do this, it must call all the destructors, but in the reverse order that they are called by the constructor. That is, the destructor starts at the top and works its way down to the base class. This is the safe and desirable thing to do because the current destructor always knows that the base-class members are alive and active since it knows what it is derived from. Thus, the destructor can perform its own cleanup, then call the next-down destructor, which will perform its own cleanup, knowing what it is derived from but not what is derived from it.

You should keep in mind that constructors and destructors are the only places where this hierarchy of calls must happen. In all other functions, only a specific function will be called, whether it's virtual or not. The only way for base-class versions of the same function to be called in ordinary functions is if you explicitly call that function.

Normally, the action of the destructor is adequate. But what happens if you want to manipulate an object through a pointer to its base class, or through its generic interface? This is certainly a major objective in object-oriented programming. The problem occurs when you want to `delete` a pointer of this type for an object that has been created on the stack with `new`. If the pointer is to the base class, the compiler can only know to call the base-class version of the destructor during `delete`. Sound familiar? This is the same problem that virtual functions were created to solve for the general case. Fortunately, as you can see in the previous example, virtual functions work for destructors as they do for all other functions except constructors.

The destructor, like the constructor, is an exceptional function, but only the destructor can be virtual because the object already knows what type it is. Once an object has been constructed, its `VTABLE` is initialized, so virtual function calls can take place.

Late binding implemented in C++ with virtual functions completes the object-oriented features of the language. It's impossible to understand or even create an example of polymorphism without using data abstraction and inheritance. Polymorphism is a feature that cannot be viewed in isolation (like `const` or a `switch` statement, for example), but instead works in concert, as part of a big picture of class relationships.

To use polymorphism and object-oriented techniques effectively in your programs, expand your view of programming to include not just members and messages of an individual class, but also the commonality between classes and their relationships with each other. Although this requires significant effort, it's a worthy struggle, since the results are faster program development, better code organization, extensible programs, and easier code maintenance. ESP

Bruce Eckel provides in-house, hands-on training seminars based on the material in this series. He is a member of the ANSI C++ committee and author of C++ Inside & Out *(Osborne/McGraw-Hill 1993; 2nd edition of* Using C++, *1989). He started as an embedded developer and published* Computer Interfacing with Pascal & C *from his columns in* Micro Cornucopia. *This material is adapted from* Thinking in C++ *(Prentice Hall, to be published in 1994). He can be reached at: Bruce Eckel, C++ Training, 20 Sunnyside Ave., Ste. A129, Mill Valley, CA 94941. Contact him by phone at (415) 331-0288, or via e-mail at 72070.3256@compuserve.com.*